Lab Manual

for

Linear Algebra

by

Jim Hefferon

Cover: my Chocolate Lab, Suzy.

Preface

WARNING! This is an incomplete draft, no doubt riddled with errors.

This collection supplements the text *Linear Algebra*[1] with explorations to help students solidify and extend their understanding of the subject, using the mathematical software Sage.[2]

A major goal of any undergraduate Mathematics program is to move students toward a higher-level, more abstract, grasp of the subject. For instance, Calculus classes work on elaborate computations while later courses spend more effort on concepts and proofs, focussing less on the details of calculations.

The text *Linear Algebra* fits into this development process. Its presentation works to bring students to a deeper understanding, but it does so by expecting that for them at this point a good bit of calculation helps the process. Naturally it uses examples and practice problems that are small-sized and have manageable numbers: an assignment to by-hand multiply a pair of three by three matrices of small integers will build intuition, whereas asking students to do that same question with twenty by twenty matrices of ten decimal place numbers would be badgering.

However, an instructor can be concerned that this misses a chance to make the point that Linear Algebra is widely applied, or to develop students's understanding through explorations that are not hindered by the mechanics of paper and pencil. Mathematical software can mitigate these concerns by extending the reach of what is reasonable to bigger systems, harder numbers, and longer computations. For instance, an advantage of learning how to handle larger jobs is that they are more like the ones that appear when students apply Linear Algebra to other areas. Another advantage is that students see new ideas such as runtime growth measures.

Well then, why not teach straight from the computer?

Our goal is to develop a higher-level understanding of the material so we want to keep the focus on vector spaces and linear maps. Our exposition takes computation to be a tool to develop that understanding, not the main object.

Some instructors will find that their students are best served by keeping a tight focus on the core material, and leaving aside altogether the work in this manual. Other instructors have students who will benefit from the increased reach that the software provides. This manual's existence, and status as a separate book, gives teachers the freedom to make the choice that suits their class.

[1] The text's home page http://joshua.smcvt.edu/linearalgebra has the PDF, the ancillary materials, and the LaTeX source.
[2] See http://www.sagemath.org for the software and documentation.

Why Sage?

In *Open Source Mathematical Software* [Joyner and Stein, 2007][1] the authors argue that for Mathematics the best way forward is to use software that is Open Source.

> Suppose Jane is a well-known mathematician who announces she has proved a theorem. We probably will believe her, but she knows that she will be required to produce a proof if requested. However, suppose now Jane says a theorem is true based partly on the results of software. The closest we can reasonably hope to get to a rigorous proof (without new ideas) is the open inspection and ability to use all the computer code on which the result depends. If the program is proprietary, this is not possible. We have every right to be distrustful, not only due to a vague distrust of computers but because even the best programmers regularly make mistakes.
>
> If one reads the proof of Jane's theorem in hopes of extending her ideas or applying them in a new context, it is limiting to not have access to the inner workings of the software on which Jane's result builds.

Professionals choose their tools by balancing many factors but this argument is persuasive. We use Sage because it is very capable so students can learn a great deal from it, and because it is Free[2] and Open Source.[3]

This manual

This is Free. Get the latest version from http://joshua.smcvt.edu/linearalgebra. Also see that page for the license details and for the LaTeX source. I am glad to get feedback, especially from instructors who have class-tested the material. My contact information is on the same page.

The computer output included here was generated automatically (I have automatically edited some lines). This is my Sage.

```
'Sage Version 5.2, Release Date: 2012-07-25'
```

Reading this manual

Here I don't define all the terms or prove all the results. So a person should read the material after covering the associated chapter in the book, using that for reference.

The association between chapters here and chapters in the book is roughly: *Python and Sage* is an introduction that does not depend on the book, *Gauss's Method* is for Chapter One, *Vector Spaces* is for Chapter Two, *Matrices*, *Maps*, and *Singular Value Decomposition* go with Chapter Three, *Geometry of Linear Maps* goes best with Chapter Four, and *Eigenvalues* fits with the material from Chapter Five (it mentions Jordan Form, but only relies on the material up to the traditional ending spot of Diagonalization.)

[1] See http://www.ams.org/notices/200710/tx071001279p.pdf for the full text. [2] The Free Software Foundation page http://www.gnu.org/philosophy/free-sw.html gives background and a definition. [3] See http://opensource.org/osd.html for a definition.

Acknowledgements

I am glad for this chance to thank the Sage Development Team. In particular, without [Sage Development Team, 2012b] this work would not have happened. I am glad also for the chance to mention [Beezer, 2011] as an inspiration. Finally, I am grateful to Saint Michael's College for the time to work on this.

We emphasize practice.
–Shunryu Suzuki [2006]

[A]n orderly presentation is not necessarily bad but by itself may be insufficient.
–Ron Brandt [1998]

Jim Hefferon
Mathematics, Saint Michael's College
Colchester, Vermont USA
2012-Sep-10

Contents

Python and Sage

To work through the Linear Algebra in this manual you need to be acquainted with how to run Sage. Sage uses the computer language Python so we'll start with that.

Python

Python is a popular computer language,[1] often used for scripting, that is appealing for its simple style and powerful libraries. The significance of 'scripting' is that Sage uses it in this way, as a glue to bring together separate parts.

Python is Free. If your operating system doesn't provide it then go to the home page www.python.org and follow the download and installation instructions. Also at that site is Python's excellent tutorial. That tutorial is thorough; here you will see only enough Python to get started. For a more comprehensive introduction see Python Team [2012b].

Comment. There is a new version, Python 3, with some differences. Here we stick to the older version because that is what Sage uses. The examples below were produced directly from Python and Sage when this manual was generated so they should be exactly what you see,[2] unless your version is quite different than mine. Here is my version of Python.

```
>>> import sys
>>> print sys.version
2.7.3 (default, Aug  1 2012, 05:16:07)
[GCC 4.6.3]
```

Basics Start Python, for instance by entering `python` at a command line. You'll get a couple of lines of information followed by three greater-than characters.

```
>>>
```

This is a prompt. It lets you experiment: if you type Python code and ⟨*Enter*⟩ then the system will read your code, evaluate it, and print the result. We will see below how to write and run whole programs but for now we will experiment. You can always leave the prompt with ⟨*Ctrl*⟩-*D*.

Try entering these expressions (double star is exponentiation).

```
>>> 2 - (-1)
3
>>> 1 + 2*3
7
```

[1]Written by a fan of Monty Python. [2]Lines that are too long are split to fit.

```
>>> 2**3
8
```

Part of Python's appeal is that doing simple things tend to be easy. Here is how you print something to the screen.

```
>>> print 1, "plus", 2, "equals", 3
1 plus 2 equals 3
```

Often you can debug just by putting in commands to print things, and having a straightforward print operator helps with that.

As in any other computer language, variables give you a place to keep values. The first line below puts one in the place called `i` and the second line uses that.

```
>>> i = 1
>>> i + 1
2
```

In some programming languages you must declare the 'type' of a variable before you use it; for instance you would have to declare that i is an integer before you could set $i = 1$. In contrast, Python deduces the type of a variable based on what you do to it — above we assigned 1 to i so Python figured that it must be an integer. Further, we can change how we use the variable and Python will go along; here we change what is in x from an integer to a string.

```
>>> x = 1
>>> x
1
>>> x = 'a'
>>> x
'a'
```

Python lets you assign multiple values simultaneously.

```
>>> first_day, last_day = 0, 365
>>> first_day
0
>>> last_day
365
```

Python computes the right side, left to right, and those values are assigned to the variables on the left. We will often use this construct.

Python complains by *raising an exception* and giving an error message. For instance, we cannot combine a string and an integer.

```
>>> 'a'+1
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

The error message's bottom line is the useful one.

Make a comment of the rest of the line with a hash mark #.

```
>>> t = 2.2
>>> d = (0.5) * 9.8 * (t**2)  # d in meters
>>> d
23.716000000000005
```

(Comments are more useful in a program than at the prompt.) Programmers often comment an entire line by starting that line with a hash.

As in the listing above, we can get real numbers and even complex numbers.[1]

```
>>> 5.774 * 3
17.322
>>> (3+2j) - (1-4j)
(2+6j)
```

As engineers do, Python uses j for the square root of -1, not i as is traditional in Mathematics.

The examples above show addition, subtraction, multiplication, and exponentiation. Division has an awkward point. Python was originally designed to have the division bar `/` mean real number division when at least one of the two numbers is real. However between two integers the division bar was taken to mean a quotient, as in “2 goes into 5 with quotient 2 and remainder 1.”

```
>>> 5.2 / 2.0
2.6
>>> 5.2 / 2
2.6
>>> 5 / 2
2
```

Experience shows this was a mistake. One of the changes in Python 3 is that the quotient operation has become `//` while the single-slash operator is always real division. In the Python 2 we are using, you must make sure that at least one number in a division is real.

```
>>> x = 5
>>> y = 2
>>> (1.0*x) / y
2.5
```

Incidentally, the integer remainder operation (sometimes called ‘modulus’) uses a percent character: `5 % 2` returns 1.

Variables can also represent truth values; these are *Booleans*.

```
>>> yankees_stink = True
>>> yankees_stink
True
```

You need the initial capital: `True` or `False`, not `true` or `false`.

Above we saw a string consisting of text between single quotes. You can use either single quotes or double quotes, as long as you use the same at both ends of the string. Here `x` and `y` are double-quoted, which makes sense because they contain apostrophes.

```
>>> x = "I'm Popeye the sailor man"
>>> y = "I yam what I yam and that's all what I yam"
>>> x + ', ' + y
"I'm Popeye the sailor man, I yam what I yam and that's all what I yam"
```

[1]Of course these aren’t actually real numbers, instead they are floating point numbers, a system that models the reals and is built into your computer’s hardware. In the prior example the distinction leaks through since its bottom line ends in 000000000005, marking where the computer’s binary approximation does not perfectly match the real that you expect. Similarly Python’s integers aren’t the integers that you studied in grade school since there is a largest one (give Python the command `import sys` followed by `sys.maxint`). However, while the issue of representation is fascinating — see Python Team [2012a] and Goldberg [1991] — we shall ignore it and just call them integers and reals.

The + operation concatenates strings. Inside a double-quoted string use slash-n \n to get a newline.

A string marked by three sets of quotes can contain line breaks.

```
>>> a = """THE ROAD TO WISDOM
...
... The road to wisdom?
... -- Well, it's plain
... and simple to express:
... Err
... and err
... and err again
... but less
... and less
... and less. --Piet Hein"""
```

The three dots at the start of lines after the first is Python's read-eval-print prompt telling you that what you have typed is not complete. We'll see below that a common use for triple-quoted strings is as documentation.

A Python *dictionary* is a finite function. That is, it is a finite set of ordered pairs $\langle \mathit{key}, \mathit{value}\rangle$ subject to the restriction that no key can be used twice. Dictionaries are a simple database.

```
>>> english_words = {'one': 1, 'two': 2, 'three': 3}
>>> english_words['one']
1
>>> english_words['four'] = 4
>>> english_words
{'four': 4, 'three': 3, 'two': 2, 'one': 1}
```

Don't be mislead by this example, the words do not always just come in the reverse of the order in which you entered them. A dictionary's elements will be listed in no apparently-sensible order.

If you assign to an existing key then that will replace the previous value.

```
>>> english_words['one'] = 5
>>> english_words
{'four': 4, 'three': 3, 'two': 2, 'one': 5}
```

Dictionaries are central to Python, in part because looking up values in a dictionary is very fast.

While dictionaries are unordered, a *list* is ordered.

```
>>> a = ['alpha', 'beta', 'gamma']
>>> b = []
>>> c = ['delta']
>>> a
['alpha', 'beta', 'gamma']
>>> a+b+c
['alpha', 'beta', 'gamma', 'delta']
```

Get an element from a list by specifying its index, its place in the list, inside square brackets. Lists are *zero-offset* indexed, that is, the initial element of the list is numbered 0. Count from the back by using negative indices.

```
>>> a[0]
'alpha'
```

```
>>> a[1]
'beta'
>>> a[-1]
'gamma'
```

Specifying two indices separated by a colon gets a *slice* of the list.

```
>>> a[1:3]
['beta', 'gamma']
>>> a[1:2]
['beta']
```

You can add to a list.

```
>>> c.append('epsilon')
>>> c
['delta', 'epsilon']
```

Lists can contain anything, including other lists.

```
>>> x = 4
>>> a = ['alpha', [True, x]]
>>> a
['alpha', [True, 4]]
```

The function `range` returns a list of numbers.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

By default `range` starts at 0, which is good because lists are zero-indexed. Observe also that 9 is the highest number in the list given by `range(10)`. This makes `range(10)+range(10,20)` give the same list as `range(20)`.

A *tuple* is like a list in that it is ordered.

```
>>> a = ('fee', 'fie', 'foe', 'fum')
>>> a
('fee', 'fie', 'foe', 'fum')
>>> a[0]
'fee'
```

However it is unlike a list in that a tuple is not *mutable*—it cannot change.

```
>>> a[0] = 'phooey'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

One reason this is useful is that because of it tuples can be keys in dictionaries while list cannot.

```
>>> a = ['Jim', 2138]
>>> b = ('Jim', 2138)
>>> d = {a: 'active'}
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

```
TypeError: unhashable type: 'list'
>>> d = {b: 'active'}
>>> d
{('Jim', 2138): 'active'}
```

Python has a special value `None` for when there is no sensible value for a variable. For instance, if your program keeps track of a person's address and includes a variable `apartment` then `None` is the right value for that variable when the person does not live in an apartment.

Flow of control Python supports the traditional ways of affecting the order of statement execution, with a twist.

```
>>> x = 4
>>> if (x == 0):
...     y = 1
... else:
...     y = 0
...
>>> y
0
```

The twist is that while many languages use braces or some other syntax to mark a block of code, Python uses indentation. (Always indent with four spaces.) Here, Python executes the single-line block `y = 1` if x equals 0, otherwise Python sets y to 0.

Notice also that double equals `==` means "is equal to." In contrast, we have already seen that single equals is the assignment operation so that `x = 4` means "x is assigned the value 4."

Python has two variants on the above `if` statement. It could have only one branch

```
>>> x = 4
>>> y = 0
>>> if (x == 0):
...     y = 1
...
>>> y
0
```

or it could have more than two branches.

```
>>> x = 2
>>> if (x == 0):
...     y = 1
... elif (x == 1):
...     y = 0
... else:
...     y = -1
...
>>> y
-1
```

Computers excel at iteration, looping through the same steps.

```
>>> for i in range(5):
...     print i, "squared is", i**2
...
```

```
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
```

A `for` loop often involves a `range`.

```
>>> x = [4, 0, 3, 0]
>>> for i in range(len(x)):
...     if (x[i] == 0):
...         print "item",i,"is zero"
...     else:
...         print "item",i,"is nonzero"
...
item 0 is nonzero
item 1 is zero
item 2 is nonzero
item 3 is zero
```

An experienced Python person who was not trying just to illustrate `range` would instead write `for c in x:` since the `for` loop can iterate over any sequence, not just a sequence of integers.

A `for` loop is designed to execute a certain number of times. The natural way to write a loop that will run an uncertain number of times is `while`.

```
>>> n = 27
>>> i = 0
>>> while (n != 1):
...     if (n%2 == 0):
...         n = n / 2
...     else:
...         n = 3*n + 1
...     i = i + 1
...     print "i=", i
...
i= 1
i= 2
i= 3
```

(This listing is incomplete; it takes 111 steps to finish.)[1] Note that "not equal" is `!=`.

The `break` command gets you out of a loop right away.

```
>>> for i in range(10):
...     if (i == 3):
...         break
...     print "i is", i
...
i is 0
i is 1
i is 2
```

A common loop construct is to run through a list performing an action on each entry. Python has a shortcut, *list comprehension*.

[1]The *Collatz conjecture* is that for any starting n this loop will terminate, but this is not known.

```
>>> a = [2**i for i in range(4)]
>>> a
[1, 2, 4, 8]
>>> [i-1 for i in a]
[0, 1, 3, 7]
```

Functions A *function* is a group of statements that executes when it is called, and can return values to the caller. Here is a naive version of the quadratic formula.

```
>>> def quad_formula(a, b, c):
...     discriminant = (b**2 - 4*a*c)**(0.5)
...     r1=(-1*b+discriminant) / (2.0*a)
...     r2=(-1*b-discriminant) / (2.0*a)
...     return (r1, r2)
...
>>> quad_formula(1,0,-9)
(3.0, -3.0)
>>> quad_formula(1,2,1)
(-1.0, -1.0)
```

(One way that it is naive is that it doesn't handle complex roots gracefully.)

Functions organize code into blocks. These blocks of code may be run a number of different times, or may belong together conceptually. In a Python program most code is in functions.

At the end of the `def` line, in parentheses, are the function's *parameters*. These can take values passed in by the caller. Functions can have *optional parameters* that have a default value.

```
>>> def hello(name="Jim"):
...     print "Hello,", name
...
>>> hello("Fred")
Hello, Fred
>>> hello()
Hello, Jim
```

Sage uses this aspect of Python heavily.

Functions always return something; if a function never executes a `return` then it will return the value `None`. They can also contain multiple `return` statements.

Objects and modules In Mathematics, the real numbers is a set associated with some operations such as addition and multiplication. Python is *object-oriented*, which means that we can similarly bundle together data and actions (in this context the functions are called *methods*).

```
>>> class DatabaseRecord(object):
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...     def salutation(self):
...         print "Dear", self.name
...
>>> a = DatabaseRecord("Jim", 53)
>>> a.name
```

```
'Jim'
>>> a.age
53
>>> a.salutation()
Dear Jim
>>> b = DatabaseRecord("Fred",109)
>>> b.salutation()
Dear Fred
```

This creates two *instances* of the object `DatabaseRecord`. The `class` code describes what these consist of. The above example uses the *dot notation*: to get the age data for the instance `a` you write `a.age`. (The `self` variable can be puzzling. It is how a method refers to the instance it is part of. Suppose that at the prompt you type `a.name="James"`. Then you've used the name `a` to refer to the instance so you can make the change. In contrast, inside the `class` description code there isn't any fixed instance. The `self` gives you a way to, for example, get the `name` attribute of the current instance.)

You won't be writing your own classes here but you will be using ones from the libraries of code that others have written, including the code for Sage, so you must know how to use the classes of others. For instance, Python has a library, or *module*, for math.

```
>>> import math
>>> math.pi
3.141592653589793
>>> math.factorial(5)
120
>>> math.cos(math.pi)
-1.0
```

The `import` statement gets the module and makes its contents available.

Programs The read-eval-print loop is great for small experiments but for more than four or five lines you want to put your work in a separate file and run it as a standalone program.

To write the code, use a text editor (one example is `Emacs`).[1] Use an editor with support for Python such as automatic indentation, and syntax highlighting, where the editor colors your code to make it easier to read.

Here is a first example of a Python program. Start your editor, open a file called `test.py`, and enter these lines. Note the triple-quoted documentation string at the top of the file; include such documentation in everything that you write.

```
# test.py
"""test

A test program for Python.
"""

import datetime

current = datetime.datetime.now()  # get a datetime object
print "the month number is", current.month
```

[1]It may come with your operating system or see http://www.gnu.org/software/emacs.

Run it through Python (for instance, from the command line run `python test.py`) and you should see output like `the month number is 9`.

Next is a small game. (It uses the Python function `raw_input` that prompts the user and then collects their response.).

```
# guessing_game.py
"""guessing_game

A toy game for demonstration.
"""
import random
CHOICE = random.randint(1,10)

def test_guess(guess):
    """Decide if the guess is correct and print a message.
    """
    if (guess < CHOICE):
        print "  Sorry, your guess is too low"
        return False
    elif (guess > CHOICE):
        print "  Sorry, your guess is too high"
        return False
    print "  You are right!"
    return True

flag = False
while (not flag):
    guess = int(raw_input("Guess an integer between 1 and 10: "))
    flag = test_guess(guess)
```

Here is the output resulting from running this game once at the command line.

```
$ python guessing_game.py
Guess an integer between 1 and 10: 5
  Sorry, your guess is too low
Guess an integer between 1 and 10: 8
  Sorry, your guess is too high
Guess an integer between 1 and 10: 6
  Sorry, your guess is too low
Guess an integer between 1 and 10: 7
  You are right!
```

As earlier, note the triple-quoted documentation strings both for the file as a whole and for the function. They provide information on how to use the guessing_game.py program as a whole, and how to use the function inside the program. Go to the directory containing guessing_game.py and start the Python read-eval-print loop. At the `>>>` prompt enter `import guessing_game`. You will play through a round of the game (there is a way to avoid this but it doesn't matter here). You are using `guessing_game` as a module. Type `help("guessing_game")`. You will see the documentation, including these lines.

```
DESCRIPTION
    A toy game for demonstration.

```

```
FUNCTIONS
    test_guess(guess)
        Decide if the guess is correct and print a message.
```

Obviously, Python got this from the file's documentation strings. In Python, and also in Sage, good practice is to always include documentation that is accessible with the `help` command. All of Sage's built-in routines do this.

Sage

Learning what Sage can do is the goal of much of this book so this is only a very brief walk-through of preliminaries. See also [Sage Development Team, 2012a] for a more broad-based introduction.

First, if your system does not already supply it then install Sage by following the directions at www.sagemath.org.

Command line Sage's command line is like Python's but adapted to mathematical work. First start Sage, for instance, enter `sage` into a command line window. You get some initial text and then a prompt (leave the prompt by typing `exit` and ⟨*Enter*⟩.)

```
sage:
```

Experiment with some expressions.

```
sage: 2**3
8
sage: 2^3
8
sage: 3*1 + 4*2
11
sage: 5 == 3+3
False
sage: sin(pi/3)
1/2*sqrt(3)
```

The second expression shows that Sage provides a convenient shortcut for exponentiation over Python's `2**3`. The fourth expression shows that Sage sometimes returns exact results rather than an approximation. You can still get the approximation; here are three equivalent ways.

```
sage: sin(pi/3).numerical_approx()
0.866025403784439
sage: sin(pi/3).n()
0.866025403784439
sage: n(sin(pi/3))
0.866025403784439
```

The function `n()` is an abbreviation for `numerical_approx()`.

Script You can group Sage commands together in a file. This way you can test the commands, and also reuse them without having to retype.

Create a file with the extension `.sage`, such as `sage_normal.sage`. Enter this function and save the file.

```
def normal_curve(upper_limit):
    """Approximate area under the Normal curve from 0 to upper_limit.
    """
    stdev = 1.0
    mu = 0.0
    area=numerical_integral((1/sqrt(2*pi) * e^(-0.5*(x)^2)),
                              0, upper_limit)
    print "area is", area[0]
```

Bring in the file with a `load` command.

```
sage: load "sage_normal.sage"
sage: normal_curve(1.0)
area is 0.341344746069
```

Notebook Sage also offers a browser-based interface, where you can set up worksheets to run alone or with other people, where you can easily view plots integrated with the text, and many other nice features.

From the Sage prompt run `notebook()` and work through the tutorial.

Gauss's Method

Sage can solve linear systems in a number of ways. The first way is to use a general system solver, one not specialized to linear systems. The second way is to leverage the special advantages of linear systems. We'll see both.

Systems

To enter a system of equations you must first enter single equations. So you must start with variables. We have seen one kind of variable in giving commands like these.

```
sage: x = 3
sage: 7*x
21
```

Here x is the name of a location to hold values. Variables in equations are something different; in the equation $C = 2\pi \cdot r$ the two variables do not have fixed values, nor are they tied to a location in the computer's memory.

To illustrate the difference enter an unassigned variable.[1]

```
sage: y
---------------------------------------------------------------------
                                                                 ----
NameError                                 Traceback (most recent call
                                                                last)

/home/ftpmaint/linear-algebra/lab/<ipython console> in <module>()

NameError: name 'y' is not defined
```

Sage defaults to expecting that y is the name of a location to hold a value. Before you use it as a symbolic variable, you must first warn the system.

```
sage: var('x,y')
(x, y)
```

[1]These output blocks are taken directly from Sage without any by-hand copying and pasting. This has the advantage that there is no chance of confusion caused by author-introduced differences between what is shown in this manual and what Sage actually does. But it has two drawbacks. First, Sage puts out some lines too long to fit in the block. Sometimes these lines will be split and have their tail carried to the next line, as here, while sometimes they will be truncated if the extra characters are unimportant. The second drawback is that Sage occasionally gives system-specific responses, as in the line here giving the directory. Naturally the response on your system will show your directory. This is the first time in this manual that the issue of these artifacts appears; we won't note them again.

```
sage: y
y
sage: 2*y
2*y
sage: k = 3
sage: 2*k
6
```

Because we haven't told Sage otherwise, it takes `k` as a location to hold values and it evaluates `2*k`. But it does not evaluate `2*y` because `y` is a symbolic variable.

With that, a system of equations is a list.

```
sage: var('x,y,z')
(x, y, z)
sage: eqns = [x-y+2*z == 4, 2*x+2*y == 12, x-4*z==5]
```

You must write double equals `==` in the equations instead of the assignment operator `=`. Also, you must write `2*x` instead of `2x`. Either mistake will trigger a `SyntaxError: invalid syntax`.

Solve the system with Sage's general-purpose solver.

```
sage: eqns = [x-y+2*z == 4, 2*x+2*y == 12, x-4*z==5]
sage: solve(eqns, x,y,z)
[[x == 5, y == 1, z == 0]]
```

You can put a parameter in the right side and solve for the variables in terms of the parameter.

```
sage: var('x,y,z,a')
(x, y, z, a)
sage: eqns = [x-y+2*z == a, 2*x+2*y == 12, x-4*z==5]
sage: solve(eqns, x,y,z)
[[x == 2/5*a + 17/5, y == -2/5*a + 13/5, z == 1/10*a - 2/5]]
```

Matrices The *solve* routine is general-purpose but for the special case of linear systems matrix notation is the best tool.

Most of the matrices in the book have entries that are fractions so here we stick to those. Sage uses 'QQ' for the rational numbers, 'RR' or 'RDF' for real numbers, 'CC' for the complex numbers, and 'ZZ' for the integers.

Enter a row of the matrix as a list, and enter the entire matrix as a list of rows.

```
sage: M = matrix(QQ, [[1, 2, 3], [4, 5, 6], [7 ,8, 9]])
sage: M
[1 2 3]
[4 5 6]
[7 8 9]
sage: M[1,2]
6
sage: M.nrows()
3
sage: M.ncols()
3
```

Sage lists are zero-indexed, as with Python lists, so `M[1,2]` asks for the entry in the second row and third column.

Enter a vector in much the same way.

```
sage: v = vector(QQ, [2/3, -1/3, 1/2])
sage: v
(2/3, -1/3, 1/2)
sage: v[1]
-1/3
```

Sage does not worry too much about the distinction between row and column vectors. Note however that it appears with rounded brackets so that it looks different than a one-row matrix.

You can augment a matrix with a vector.

```
sage: M = matrix(QQ, [[1, 2, 3], [4, 5, 6], [7 ,8, 9]])
sage: v = vector(QQ, [2/3, -1/3, 1/2])
sage: M_prime = M.augment(v)
sage: M_prime
[   1    2    3  2/3]
[   4    5    6 -1/3]
[   7    8    9  1/2]
```

You can use an optional argument to have Sage remember the distinction between the two parts of M'.

```
sage: M_prime = M.augment(v,subdivide=True)
sage: M_prime
[   1    2    3| 2/3]
[   4    5    6|-1/3]
[   7    8    9| 1/2]
```

Row operations Computers are good for jobs that are tedious and error-prone. Row operations are both.

```
sage: M = matrix(QQ, [[0, 2, 1], [2, 0, 4], [2 ,-1/2, 3]])
sage: v = vector(QQ, [2, 1, -1/2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[   0    2    1|   2]
[   2    0    4|   1]
[   2 -1/2    3|-1/2]
```

Swap the top rows (remember that row indices start at zero).

```
sage: M_prime.swap_rows(0,1)
sage: M_prime
[   2    0    4|   1]
[   0    2    1|   2]
[   2 -1/2    3|-1/2]
```

Rescale the top row.

```
sage: M_prime.rescale_row(0, 1/2)
sage: M_prime
[   1    0    2| 1/2]
[   0    2    1|   2]
[   2 -1/2    3|-1/2]
```

Get a new bottom row by adding -2 times the top to the current bottom row.

```
sage: M_prime.add_multiple_of_row(2,0,-2)
sage: M_prime
[   1     0     2|  1/2]
[   0     2     1|    2]
[   0  -1/2    -1| -3/2]
```

Finish by finding echelon form.

```
sage: M_prime.add_multiple_of_row(2,1,1/4)
sage: M_prime
[   1     0     2|  1/2]
[   0     2     1|    2]
[   0     0  -3/4|   -1]
```

By the way, Sage would have given us echelon form in a single operation had we run the command `M_prime.echelon_form()`.

Now by-hand back substitution gives the solution, or we can use `solve`.

```
sage: var('x,y,z')
(x, y, z)
sage: eqns=[-3/4*z == -1, 2*y+z == 2, x+2*z == 1/2]
sage: solve(eqns, x, y, z)
[[x == (-13/6), y == (1/3), z == (4/3)]]
```

The operations `swap_rows`, `rescale_rows`, and `add_multiple_of_row` changed the matrix M'. Sage has related commands that return a changed matrix but leave the starting matrix unchanged.

```
sage: M = matrix(QQ, [[1/2, 1, -1], [1, -2, 0], [2 ,-1, 1]])
sage: v = vector(QQ, [0, 1, -2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1/2    1   -1|   0]
[  1   -2    0|   1]
[  2   -1    1|  -2]
sage: N = M_prime.with_rescaled_row(0,2)
sage: M_prime
[1/2    1   -1|   0]
[  1   -2    0|   1]
[  2   -1    1|  -2]
sage: N
[ 1   2  -2| 0]
[ 1  -2   0| 1]
[ 2  -1   1|-2]
```

Here, M' is unchanged by the routine while N is the returned changed matrix. The other two routines of this kind are `with_swapped_rows` and `with_added_multiple_of_row`.

Nonsingular and singular systems Resorting to `solve` after going through the row operations is artless. Sage will give reduced row echelon form straight from the augmented matrix.

```
sage: M = matrix(QQ, [[1/2, 1, -1], [1, -2, 0], [2 ,-1, 1]])
```

```
sage: v = vector(QQ, [0, 1, -2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime.rref()
[      1       0       0|   -4/5]
[      0       1       0|  -9/10]
[      0       0       1|-13/10]
```

In that example the matrix M on the left is nonsingular because it is square and because Gauss's Method produces echelon forms where every one of M's columns has a leading variable. The next example starts with a different square matrix, a singular matrix, and consequently leads to echelon form systems where there are columns on the left that do not have a leading variable.

```
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: v = vector(QQ, [0, 1, 1])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1|0]
[1 2 3|1]
[2 3 4|1]
sage: M_prime.rref()
[ 1  0 -1|-1]
[ 0  1  2| 1]
[ 0  0  0| 0]
```

Recall that the singular case has two subcases. The first is above: because in echelon form every row that is all zeros on the left has an entry on the right that is also zero, the system has infinitely many solutions. In contrast, with the same starting matrix the example below has a row that is zeros on the left but is nonzero on the right and so the system has no solution.

```
sage: v = vector(QQ, [0, 1, 2])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime.rref()
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 1]
```

The difference between the subcases has to do with the relationships among the rows of M and the relationships among the rows of the vector. In both cases, the relationship among the rows of the matrix M is that the first two rows add to the third. In the first case the vector has the same relationship while in the second it does not.

The easy way to ensure that a zero row in the matrix on the left is associated with a zero entry in the vector on the right is to make the vector have all zeros, that is, to consider the homogeneous system associated with M.

```
sage: v = zero_vector(QQ, 3)
sage: v
(0, 0, 0)
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1|0]
[1 2 3|0]
[2 3 4|0]
```

```
sage: M_prime.rref()
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 0]
```

You can get the numbers of the columns having leading entries with the `pivots` method (there is a complementary `nonpivots`).

```
sage: M_prime
[1 1 1|0]
[1 2 3|1]
[2 3 4|2]
sage: M_prime.pivots()
(0, 1, 3)
sage: M_prime.rref()
[ 1  0 -1| 0]
[ 0  1  2| 0]
[ 0  0  0| 1]
```

Because column 2 is not in the list of pivots we know that the system is singular before we see it in echelon form.

We can use this observation to write a routine that decides if a square matrix is nonsingular.

```
sage: def check_nonsingular(mat):
....:     if not(mat.is_square()):
....:         print "ERROR: mat must be square"
....:         return
....:     p = mat.pivots()
....:     for col in range(mat.ncols()):
....:         if not(col in p):
....:             print "nonsingular"
....:             break
....:
sage: N = Matrix(QQ, [[1, 2, 3], [4, 5, 6], [7, 8, 9]])
sage: check_nonsingular(N)
nonsingular
sage: N = Matrix(QQ, [[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: check_nonsingular(N)
```

Actually, Sage matrices already have a method `is_singular` but this illustrates how you can write routines to extend Sage.

Parametrization You can use `solve` to give the solution set of a system with infinitely many solutions. Start with the square matrix of coefficients from above, where the top two rows add to the bottom row, and adjoin a vector satisfying the same relationship to get a system with infinitely many solutions. Then convert that to a system of equations.

```
sage: M = matrix(QQ, [[1, 1, 1], [1, 2, 3], [2 , 3, 4]])
sage: v = vector(QQ, [1, 0, 1])
sage: M_prime = M.augment(v, subdivide=True)
sage: M_prime
[1 1 1|1]
[1 2 3|0]
```

```
[2 3 4|1]
sage: var('x,y,z')
(x, y, z)
sage: eqns = [x+y+z == 1, x+2*y+3*z == 0, 2*x+3*y+4*z == 1]
sage: solve(eqns, x, y)
[[x == z + 2, y == -2*z - 1]]
sage: solve(eqns, x, y, z)
[[x == r1 + 2, y == -2*r1 - 1, z == r1]]
```

The first of the two `solve` calls asks Sage to solve only for x and y and so the solution is in terms of z. In the second call Sage produces a parameter of its own.

Automation

We finish by showing two routines to automate the by-hand row reductions of the kind that the text has in the homework. These use the matrix capabilities of Sage to both describe and perform the row operations that bring a matrix to echelon form or to reduced echelon form.

Loading and running The source file of the script is below, at the end. First here are a few sample calls. Start Sage in the directory containing the file `gauss_method.sage`.

```
sage: load "gauss_method.sage"
sage: M = matrix(QQ, [[1/2, 1, 4], [2, 4, -1], [1, 2, 0]])
sage: v = vector(QQ, [-2, 5, 4])
sage: M_prime = M.augment(v, subdivide=True)
sage: gauss_method(M_prime)
[1/2   1   4| -2]
[  2   4  -1|  5]
[  1   2   0|  4]
take -4 times row 1 plus row 2
take -2 times row 1 plus row 3
[1/2   1   4| -2]
[  0   0 -17| 13]
[  0   0  -8|  8]
take -8/17 times row 2 plus row 3
[  1/2     1     4|   -2]
[    0     0   -17|   13]
[    0     0     0|32/17]
```

Because the matrix has rational number elements the operations are exact — without floating point issues.

The remaining examples skip the steps to make an augmented matrix.

```
sage: M1 = matrix(QQ, [[2, 0, 1, 3], [-1, 1/2, 3, 1], [0, 1, 7, 5]])
sage: gauss_method(M1)
[  2   0   1   3]
[ -1 1/2   3   1]
[  0   1   7   5]
take 1/2 times row 1 plus row 2
[  2   0   1   3]
[   0 1/2 7/2 5/2]
[   0   1   7   5]
take -2 times row 2 plus row 3
[   2   0   1   3]
[   0 1/2 7/2 5/2]
[   0   0   0   0]
```

The script also has a routine to go all the way to reduced echelon form.

```
sage: r1 = [1, 2, 3, 4]
sage: r2 = [1, 2, 3, 4]
sage: r3 = [2, 4, -1, 5]
sage: r4 = [1, 2, 0, 4]
sage: M2 = matrix(QQ, [r1, r2, r3, r4])
sage: gauss_jordan(M2)
[ 1  2  3  4]
[ 1  2  3  4]
[ 2  4 -1  5]
[ 1  2  0  4]
take -1 times row 1 plus row 2
take -2 times row 1 plus row 3
take -1 times row 1 plus row 4
[ 1  2  3  4]
[ 0  0  0  0]
[ 0  0 -7 -3]
[ 0  0 -3  0]
swap row 2 with row 3
[ 1  2  3  4]
[ 0  0 -7 -3]
[ 0  0  0  0]
[ 0  0 -3  0]
take -3/7 times row 2 plus row 4
[   1   2   3   4]
[   0   0  -7  -3]
[   0   0   0   0]
[   0   0   0 9/7]
swap row 3 with row 4
[   1   2   3   4]
[   0   0  -7  -3]
[   0   0   0 9/7]
[   0   0   0   0]
take -1/7 times row 2
take 7/9 times row 3
[   1   2   3   4]
[   0   0   1 3/7]
[   0   0   0   1]
[   0   0   0   0]
take -4 times row 3 plus row 1
take -3/7 times row 3 plus row 2
[1 2 3 0]
[0 0 1 0]
[0 0 0 1]
```

```
[0 0 0 0]
take -3 times row 2 plus row 1
[1 2 0 0]
[0 0 1 0]
[0 0 0 1]
[0 0 0 0]
```

These are naive implementations of Gauss's Method that are just for fun. (For instance they don't handle real numbers, just rationals.) But they do illustrate the point made in this manual's Preface, since a person builds intuition by doing a reasonable number of reasonably hard Gauss's Method reductions by hand, and at that point automation can take over.

Source of `gauss_method.sage` *Code comment:* lines 50 and 88 are too long for this page so they end with a slash \ to make Python continue on the next line.

```
1  # code gauss_method.sage
2  # Show Gauss's method and Gauss-Jordan reduction steps.
3  # 2012-Apr-20  Jim Hefferon  Public Domain.
4  
5  # Naive Gaussian reduction
6  def gauss_method(M,rescale_leading_entry=False):
7      """Describe the reduction to echelon form of the given matrix of
8      rationals.
9  
10     M  matrix of rationals   e.g., M = matrix(QQ, [[..], [..], ..])
11     rescale_leading_entry=False  boolean  make leading entries to 1's
12 
13     Returns: None.  Side effect: M is reduced.  Note that this is
14     echelon form, not reduced echelon form;, and that this routine
15     does not end the same way as does M.echelon_form().
16 
17     """
18     num_rows=M.nrows()
19     num_cols=M.ncols()
20     print M
21 
22     col = 0   # all cols before this are already done
23     for row in range(0,num_rows):
24         # ?Need to swap in a nonzero entry from below
25         while (col < num_cols
26                and M[row][col] == 0):
27       for i in M.nonzero_positions_in_column(col):
28           if i > row:
29                     print " swap row",row+1,"with row",i+1
30         M.swap_rows(row,i)
31                     print M
32                     break
33             else:
34                 col += 1
35 
36   if col >= num_cols:
37             break
```

```

        # Now guaranteed M[row][col] != 0
        if (rescale_leading_entry
           and M[row][col] != 1):
            print " take",1/M[row][col],"times row",row+1
            M.rescale_row(row,1/M[row][col])
      print M
        change_flag=False
        for changed_row in range(row+1,num_rows):
            if M[changed_row][col] != 0:
                change_flag=True
                factor=-1*M[changed_row][col]/M[row][col]
                print " take",factor,"times row",row+1,  \
                      "plus row",changed_row+1
                M.add_multiple_of_row(changed_row,row,factor)
        if change_flag:
            print M
        col +=1

# Naive Gauss-Jordan reduction
def gauss_jordan(M):
    """Describe the reduction to reduced echelon form of the
    given matrix of rationals.

    M  matrix of rationals   e.g., M = matrix(QQ, [[..], [..], ..])

    Returns: None.  Side effect: M is reduced.

    """
    gauss_method(M,rescale_leading_entry=False)
    # Get list of leading entries [le in row 0, le in row1, ..]
    pivot_list=M.pivots()
    # Rescale leading entries
    change_flag=False
    for row in range(0,len(pivot_list)):
        col=pivot_list[row]
        if M[row][col] != 1:
            change_flag=True
            print " take",1/M[row][col],"times row",row+1
            M.rescale_row(row,1/M[row][col])
    if change_flag:
        print M
    # Pivot
    for row in range(len(pivot_list)-1,-1,-1):
        col=pivot_list[row]
        change_flag=False
        for changed_row in range(0,row):
            if M[changed_row,col] != 0:
                change_flag=True
                factor=-1*M[changed_row][col]/M[row][col]
                print " take",factor,"times row",   \
                              row+1,"plus row",changed_row+1
                      M.add_multiple_of_row(changed_row,row,factor)
           if change_flag:
                print M
```

Bibliography

Robert A. Beezer. Sage for Linear Algebra. http://linear.ups.edu/download/fcla-2.22-sage-4.7.1-preview.pdf, 2011.

David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991.

Jim Hefferon. Linear Algebra. http://joshua.smcvt.edu/linearalgebra, 2012.

David Joyner and William Stein. Open source mathematical software. *Notices of the AMS*, page 1279, November 2007.

Python Team. Floating point arithmetic: issues and limitations, 2012a. URL http://docs.python.org/2/tutorial/floatingpoint.html. [Online; accessed 17-Dec-2012].

Python Team. The python tutorial, 2012b. URL http://docs.python.org/2/tutorial/index.html. [Online; accessed 17-Dec-2012].

Ron Brandt. *Powerful Learning*. Association for Supervision and Curriculum Development, 1998.

Sage Development Team. Sage tutorial 5.3. http://www.sagemath.org/pdf/SageTutorial.pdf, 2012a.

Sage Development Team. Sage reference manual 5.3. http://www.sagemath.org/pdf/reference.pdf, 2012b.

Shunryu Suzuki. *Zen Mind, Beginners Mind*. Shambhala, 2006.